DIVISION DE EDUCACION CONTINUA
FACULTAD DE INGENIERIA, UNAM
CURSOS ABIERTOS

DIVISIÓN DE EDUCACIÓN CONTINUA

CURSO: ***CC043 Visual Basic Avanzado***
FECHA: ***2 al 13 de julio del 2001***

EVALUACIÓN DEL PERSONAL DOCENTE

(ESCALA DE EVALUACIÓN: 1 A 10)

CONFERENCISTA	DOMINIO DEL TEMA	USO DE AYUDAS AUDIOVISUALES	COMUNICACIÓN CON EL ASISTENTE	PUNTUALIDAD
TEC. YAZMIN VEGA HERNANDEZ				

Promedio _____

EVALUACIÓN DE LA ENSEÑANZA

CONCEPTO	CALIF.
ORGANIZACIÓN Y DESARROLLO DEL CURSO	
GRADO DE PROFUNDIDAD DEL CURSO	
ACTUALIZACION DEL CURSO	
APLICACIÓN PRACTICA DEL CURSO	

Promedio _____

EVALUACIÓN DEL CURSO

CONCEPTO	CALIF.
CUMPLIMIENTO DE LOS OBJETIVOS DEL CURSO	
CONTINUIDAD EN LOS TEMAS	
CALIDAD DEL MATERIAL DIDÁCTICO UTILIZADO	

Promedio _____

Evaluación total del curso_____

Continúa...2

1. ¿Le agradó su estancia en la División de Educación Continua?

SI ☐ NO ☐

Si indica que "NO" diga porqué: _____

2. Medio a través del cual se enteró del curso:

Periódico *La Jornada*	
Folleto anual	
Folleto del curso	
Gaceta UNAM	
Revistas técnicas	
Otro medio (Indique cuál)	

3 ¿Qué cambios sugeriría al curso para mejorarlo?

_____

_____

_____

_____

_____

_____

4. ¿Recomendaría el curso a otra(s) persona(s) ?

SI ☐ NO ☐

5.¿Qué cursos sugiere que imparta la División de Educación Continua?

_____

_____

_____

_____

_____

_____

6. Otras sugerencias.

_____

_____

_____

_____

_____

FACULTAD DE INGENIERÍA UNAM
DIVISIÓN DE EDUCACIÓN CONTINUA

"Tres décadas de orgullosa excelencia" 1971 - 2001

VISUAL BASIC AVANZADO

JULIO DEL 2001

Palacio de Minería, Calle de Tacuba No 5, Primer piso, Delegación Cuauhtémoc, CP 06000, Col Centro, México D F,
APDO Postal M-2285 • Tels: 5521 4021 al 24, 5623.2910 y 5623 2971 • Fax: 5510 0573

1. FUNDAMENTOS DE PROGRAMACIÓN ORIENTADA A OBJETOS. (EL ENFOQUE DE VISUAL BASIC.)

1.1 Conceptos Básicos.

1.1 Mecanismos básicos.

Los mecanismos básicos de la orientación a objetos son los objetos, mensajes y los métodos, clases y variables instancia (o modelo) y herencia. Todos los sistemas que merecen la descripción de orientado a objetos contienen estos mecanismos esenciales, aunque los mecanismos pueden no estar realizados (o denominados) exactamente de la misma forma.

1.1.1 Objetos.

Se definirá un objeto como un concepto, abstracción o cosa con limites bien definidos y con significado a efectos del problema que se tenga entre manos. Los objetos tienen dos propósitos: promover la comprensión del mundo real y proporcionar una base practica para la implementación por computadora. La descomposición de un problema en objetos depende del juicio y de la naturaleza del problema. No existe una única representación correcta.

Un programa tradicional consta de procedimientos y datos. Un programa orientado a objetos consta solamente de objetos que contienen tanto los procedimientos como los datos. Por decirlo de otra forma, los objetos son módulos que contienen los datos y las instrucciones que operan sobre esos datos. Así, dentro de los objetos residen los datos de los lenguajes convencionales, como por ejemplo, números, matrices (arrays o arreglos), cadenas de caracteres y registros, así como cualquier función, instrucción o subrutina que opere sobre ellos. Los objetos, por tanto, son entidades que tienen atributos (datos) y formas de comportamiento (procedimientos) particulares.

Los objetos llevan los nombres de los elementos de interés desde el dominio

de la aplicación. Por ejemplo, en una aplicación de procesamiento de textos, es probable que un objeto sea llamado párrafo. En una aplicación de contabilidad, la hoja balance de junio es un objeto probable. Desde la perspectiva del usuario, los objetos proporcionan el comportamiento deseado. Un párrafo puede aceptar revisión y realinear sus márgenes. La hoja balance del mes de junio puede imprimirse o consolidarse con las de otros meses para constituir un informe cuatrimestral. Desde la perspectiva del programador, los objetos son módulos de una aplicación que funcionan juntos para proporcionar una funcionalidad general.

Es importante señalar que todos los objetos poseen su propia identidad y se pueden distinguir entre si. Dos manzanas del mismo color, forma y textura siguen siendo manzanas individuales. El termino identidad significa que los objetos se distinguen por su existencia inherente y no por las propiedades descriptivas que puedan tener.

Las aplicaciones pueden constar de diferentes clases de objetos. Un objeto activo es aquel que se comprende su propio hilo de control, mientras que un objeto pasivo no. Los objetos activos suelen ser autónomos lo que quiere decir que pueden exhibir algún comportamiento sin que ningún otro objeto opere sobre ellos. Los objetos pasivos, por otra parte, solo pueden padecer un cambio de estado cuando se actúa explícitamente sobre ellos. De este modo los objetos activos de un sistema sirven como raíces de control.

1.1.2 Mensajes y Métodos.

En la mayoría de los lenguajes de programación orientados a objetos las operaciones que los clientes pueden realizar sobre un objeto suelen declararse como métodos, que forman parte de la declaración de la clase. El paso de mensajes es una de las partes de la ecuación que define el comportamiento de un objeto; la definición de comportamiento también recoge que el estado de un objeto afecta a si mismo a su comportamiento. A diferencia de los elementos de datos pasivos en los sistemas tradicionales, los objetos tienen la posibilidad de actuar. La acción sucede cuando un objeto recibe un mensaje, que es, una solicitud que

pide al objeto que se comporte de alguna forma. Cuando se ejecutan los programas orientados a objetos, los objetos reciben, interpretan y responden a mensajes procedentes de los objetos. Por ejemplo, cuando un usuario solicita que un objeto llamado documento se imprima a sí mismo, el documento puede enviar un mensaje al objeto impresora solicitando un lugar en la cola de impresión; el objeto impresora puede devolver un mensaje al documento solicitando información de formato, y así sucesivamente. Los mensajes pueden contener información para clarificar una solicitud; por ejemplo, el mensaje solicitando que un objeto se imprima a sí mismo podría incluir el nombre de la impresora. Finalmente, el emisor del mensaje no necesita conocer la forma en que el objeto receptor está llevando a cabo la solicitud. En otras palabras, cuando el objeto documento recibe el mensaje imprimir, documento sabe exactamente lo que tiene qué hacer. El objeto que envía el mensaje ni sabe ni le importa cómo se realiza la impresión, solamente conoce que está sucediendo.

El conjunto de mensajes al que un objeto puede responder se llama protocolo del objeto. El protocolo para un icono puede constar de mensajes invocados por la pulsación del botón del ratón cuando el usuario localiza un puntero sobre un icono.

Los métodos pueden enviar también mensajes a otros objetos solicitando acción o información.

Al igual que las "cajas negras" de la ingeniería, la estructura interior de un objeto está oculta a usuarios y programadores. Los mensajes que recibe el objeto son los únicos contactos que conectan al objeto con el mundo exterior. Solamente un método del propio objeto puede disponer de los datos del interior del mismo para su manipulación. Estas características de los objetos confieren a la orientación a objetos su ventaja: La orientación a objetos fomenta la modularidad haciendo muy claras las fronteras entre objetos, explícita la comunicación entre los mismos y ocultos los detalles de la realización.

Cuando se ejecuta un programa orientado a objetos, ocurren tres sucesos. En primer lugar, se crean los objetos cuando se necesitan. Segundo, los mensajes se mueven de un objeto a otro (o desde él usuario a un objeto) a medida que el programa procesa internamente información o responde a la entrada del usuario.

Finalmente, se borran los objetos cuando ya no son necesarios y se recupera memoria.

1.1.3 Clases, subclases y objetos.

Muchos objetos diferentes pueden actuar de formas muy similares. Una clase es una abstracción que describe propiedades importantes para una aplicación y que ignora el resto. Una clase consta de métodos y datos que resumen las características comunes de un conjunto de objetos. La posibilidad de abstraer métodos y descripciones de datos comunes de un conjunto de objetos y almacenarlos en una clase es esencial para la potencia de la orientación a objetos . Definir clases significa situar código reutilizable en un depósito común en lugar de volver a expresarlo una vez y otra. En otras palabras, las clases contienen los anteproyectos para crear objetos. Finalmente, la definición de una clase ayuda a clarificar la definición de un objeto: un objeto es un modelo o instancia de una clase.

Los objetos se crean cuando se recibe un mensaje solicitando creación por la clase padre. El nuevo objeto toma sus métodos y datos de su clase padre. Los datos son de dos formas, variables de clase y variables modelo o de instancia. Las variables de clase tienen valores almacenados en una clase; las variables instancia tienen valores asociados únicamente con cada instancia u objeto creado a partir de una clase.

A título de ejemplo, considere cómo un programador podría designar una aplicación de procesamiento de textos en forma orientada a objetos. En primer lugar el programador identifica las entidades de interés. Los párrafos, por ejemplo, son objetos potenciales. Justificar es un método común para todos los párrafos. Tipodeletra (Fuente) es una variable de clase con el valor helvética. Finalmente, texto es una variable modelo con valores únicos para cada objeto. Es útil crear una clase llamada párrafo para guardar esta información común. La clase párrafo proporciona entonces un anteproyecto para la construcción de objetos. Aunque los datos específicos de cada objeto (ej.: las palabras del párrafo, el tipo de letra o

fuente, el interlineado o espaciado) pueden variar, todos los objetos de la clase párrafo comparten métodos y variables de clase comunes.

Una clase puede también resumir elementos comunes para un conjunto de subclases. En el caso del ejemplo del procesamiento de textos, el programador puede considerar después tabla para que sea otra clase además de párrafo. Pensando un poco más, el programador se da cuenta que párrafo y tabla comparten algunas propiedades con una clase más abstracta, texto. De forma similar, texto y gráfico pueden convertirse en subclases de una clase con carácter aún más general, documento ilustra esta jerarquía de las clases del procesamiento del documento. Utilizando subclases, los programadores orientados a objetos describen las aplicaciones como conjuntos de módulos generales o abstractos. Los métodos y datos comunes se elevan tan alto como sea posible de forma que sean accesibles a todas las subclases relacionadas.

A veces se denomina a las subclases como clases derivadas. En otras ocasiones los términos padre e hija se utilizan para indicar la relación entre una clase y una subclase. Las clases padre están localizadas por encima de las clases hijas en la jerarquía. Las clases más altas en la jerarquía se denominan las superclases.

Hasta ahora, esta descripción de un diseño orientado a objetos ha sido "ascendente". Es decir, la descripción pone el énfasis en la forma en que los componentes (objetos) son abstraídos para llegar a ser clases y superclases. De hecho, el término "descendente" ("top-down") describe con mayor precisión el método seguido por muchos programadores de la orientación a objetos. Normalmente comienzan su trabajo con una biblioteca de clases que contiene generalmente módulos útiles de programación. Utilizando clases predefinidas como punto inicial, los programadores escriben nuevas subclases que adaptan las clases de empleo general de la biblioteca a los requisitos funcionales particulares de la aplicación.

Una biblioteca de clases específicas para una aplicación se denomina un marco estructural "framework". Los marcos estructurales difieren de las bibliotecas de clases en su distinto grado: un marco estructural es una biblioteca de clases

ajustada especialmente para una determinada categoría de aplicaciones, por ejemplo, un marco estructural constructor de interfaces. Construir y adaptar aplicaciones a partir de marcos estructurales es más rápido y fácil que empezar con bibliotecas de clases genéricas. Asimismo, un marco estructural no será normalmente útil fuera del campo de la aplicación ya que contiene clases específicas para la aplicación.

1.1.4 Herencia.

La herencia es el mecanismo para compartir automáticamente métodos y datos entre clases, subclases y objetos. En términos generales se puede definir una clase que después se ira refinando sucesivamente para producir subclases. Todas las subclases poseen, o heredan, todas y cada una de las propiedades de su superclase y añaden además, sus propiedades exclusivas. No es necesario repetir las propiedades de las superclases en cada subclase. La herencia permite a los programadores crear nuevas clases programando solamente las diferencias con la clase padre. Cuando un programador declara que párrafo es una subclase de texto, por ejemplo, todos los métodos y variables modelo asociados con texto son heredados automáticamente por párrafo. Si la clase texto contiene métodos que son inadecuados para la subclase párrafo, entonces el programador puede obviar estos métodos escribiendo unos nuevos y almacenándolos como parte de la clase párrafo.

Debido a la herencia, los programas orientados a objetos constan de taxonomías, árboles o jerarquías de clases que, por medio de la sub clasificación, llegan a ser más específicas. Las clases proporcionan los anteproyectos para las subclases o para los objetos relacionados con una aplicación.

Herencia simple y múltiple son dos tipos de mecanismos de herencia utilizados normalmente en la programación orientada a objetos. Con la herencia simple, una subclase puede heredar datos y métodos de una clase simple así como añadir o sustraer comportamiento por sí misma. La herencia múltiple se refiere a la posibilidad de una subclase de adquirir los datos y métodos de más de

una clase. La herencia múltiple es útil al construir comportamiento compuesto a partir de más de una rama de una jerarquía de clases.

En resumen, los mecanismos básicos de la orientación a objetos conducen a una particular visión sobre el concepto de dar forma al mundo. Los elementos y su comportamiento se identifican como objetos. El comportamiento se realiza con métodos y datos almacenados en el objeto. Los mensajes obtienen el comportamiento de un objeto invocando un método del mismo.

Los objetos con métodos y variables modelo comunes se reúnen en una clase. Las clases se organizan en jerarquías y los mecanismos de herencia proporcionan automáticamente a cada subclase los métodos y datos de las clases padre. Las subclases se crean programando las diferencias entre las clases disponibles en una librería y los requisitos particulares de la aplicación.

1.2 Conceptos clave.

Los mecanismos básicos señalados anteriormente forman la base del paradigma de la orientación a objetos. Cuatro conceptos clave que resumen las ventajas del método orientado a objetos son la encapsulación, la abstracción, el polimorfismo y la persistencia.

1.2.1 Encapsulación.

La encapsulación (denominada también ocultamiento de información) consiste en separar los aspectos externos del objeto, a los cuales pueden acceder otros objetos, de los detalles internos de implementación del mismo, que quedan ocultos para los demás. La encapsulación evita que el programa llegue a ser tan interdependiente que un pequeño cambio tenga efectos secundarios masivos. La implementación de un objeto se puede modificar sin afectar a las aplicaciones que la utilizan. Quizá sea necesario modificar la implementación de un objeto para mejorar el rendimiento, corregir un error, consolidar el código o para hacer un

transporte a otra plataforma.

La encapsulación no es exclusiva de la programación orientada a objetos pero la capacidad de combinar la estructura de datos y el comportamiento en una única entidad hace que la encapsulación sea aquí mas limpia y potente que en los lenguajes convencionales que separan las estructuras de datos y el comportamiento.

1.2.2 Abstracción.

La abstracción consiste en centrarse en los aspectos esenciales inherentes de una entidad, e ignorar sus propiedades accidentales. En el desarrollo de sistemas esto significa centrarse en lo que es y lo que hace un objeto antes de decidir como debería ser implementado. El uso de la abstracción mantiene nuestra libertad de tomar decisiones durante el mayor tiempo posible evitado comprometernos de forma prematura con ciertos detalles. La mayoría de los lenguajes modernos proporcionan abstracción de datos pero la capacidad de utilizar herencia y polimorfismo proporciona una potencia adicional. El uso de la abstracción durante el análisis significa tratar solamente conceptos del dominio de la aplicación y no tomar decisiones de diseño o de implementación antes de haber comprendido el problema. Un uso adecuado de la abstracción permite utilizar el mismo modelo para el análisis, diseño de alto nivel, estructura del programa, estructura de una base de datos y documentación. Un estilo de diseño independiente del lenguaje pospone los detalles de programación hasta la fase final, relativamente mecánica del desarrollo.

1.2.3 Polimorfismo.

Los objetos actúan en respuesta a los mensajes que reciben. El mismo mensaje puede originar acciones completamente diferentes al ser recibido por diferentes objetos. Este fenómeno se conoce como polimorfismo. Con el polimorfismo un usuario puede enviar un mensaje genérico y dejar los detalles

exactos de la realización para el objeto receptor. El mensaje imprimir, por ejemplo, al ser enviado a una figura o diagrama invocará diferentes métodos de impresión que en el caso de enviar el mismo mensaje imprimir a un documento de texto.

El polimorfismo está fomentado por la maquinaria de la herencia. Es muy normal almacenar los protocolos para funciones de utilidad como "imprimir" en la posición más alta que sea posible dentro de la jerarquía de clases. Las variaciones necesarias en el comportamiento "imprimir" se almacenan en niveles más bajos de la jerarquía para sobrescribir los métodos más generales cuando sea necesario. De esta forma, los objetos están listos y pueden responder apropiadamente a mensajes de utilidad como "imprimir" mientras que el método que realiza la función de impresión puede existir en la clase inmediata del objeto o varios niveles por encima de la clase del objeto.

1.2.4 Persistencia.

La persistencia se refiere a la permanencia de un objeto, es decir, al tiempo durante el cual se asigna espacio y permanece accesible en la memoria de la computadora. En la mayoría de los lenguajes orientados a objetos, se crean modelos de clases a medida que el programa se ejecuta. Algunos de estos modelos se necesitan solamente por un breve período de tiempo. Cuando un objeto ya no es necesario, es destruido y recuperado el espacio de memoria que tenía asignado. La recuperación automática del espacio de memoria se denomina normalmente recolección de basura.

Después de haber ejecutado un programa orientado a objetos, los objetos ensamblados normalmente no se almacenan; es decir, los objetos dejan de ser persistentes. Una base de datos orientada a objetos mantiene una distinción entre objetos creados solamente para el tiempo de duración de la ejecución y aquellos pensados para almacenamiento permanente. Los objetos almacenados permanentemente se denominan persistentes.

1.3 La POO en Visual Basic.

Visual Basic soporta algunos de los mecanismos básicos de la programación orientada a objetos, y algunos otros los incluye modificados para adaptarse al modelo bajo el que se encuentra diseñado el lenguaje.

En VB Los objetos también se encuentran encapsuladoses decir, contienen sus *propiedades, métodos* y en el modelo de VB también *eventos.* Las propiedades como en el caso general son los datos que describen un objeto. Los eventos son acciones que pueden ser reconocidas por un objeto (un clic sobre una opción en un menú es un evento que produce una acción). Un método en este caso agrupa el código de la acción que se ejecuta en respuesta a un evento.

Al conjunto de propiedades y métodos de una clase recibe el nombre de *interfaz.* Los objetos, además de su interfaz predeterminada, pueden implementar interfaces adicionales para proporcionar polimorfismo. La mayoría de los lenguajes orientados a objetos proporcionan polimorfismo mediante la herencia. Visual Basic utiliza otra técnica llamada de interfaz múltiple del modelo de objetos componente (COM) que permite que los programas crezcan con el tiempo, agregando nuevas funcionalidades sin afectar las ya existentes. El polimorfismo permite manipular muchos tipos diferentes de objetos sin preocuparse de su tipo.

Los objetos de Visual Basic también se crean a partir de *clases.* La clase define las interfaces de un objeto, el alcance del objeto, y las restricciones para que este pueda ser utilizado. Las descripciones de las clases se almacenan en *bibliotecas de tipos* que pueden examinarse en el *examinador de objetos.*

Para utilizar un objeto hay que asignarle una *referencia* a una *variable objeto.*

MÓDULOS DE CLASE

En Visual Basic las clases de objetos se definen en *módulos de clase.* En ellos se codifican los *métodos,* las *propiedades* que definen las características comunes a todos los objetos de esa clase y los *eventos* que puede desencadenar un objeto de esa clase, a los que el programa podrá responder. La clave de la programación orientada a objetos está en la calidad de la abstracción de los métodos y las propiedades de los objetos que serán englobados en una clase. Los *métodos,* las *propiedades* y los *eventos* reciben en su conjunto el nombre de *miembros* de la clase.

Cada módulo de clase contiene la definición de una sola *clase.* Los *métodos* de la clase son los procedimientos y las funciones definidas en el módulo, las *propiedades* las variables definidas a nivel del módulo o los procedimientos **Property**, y los *eventos* serán las declaraciones de tipo **Event.**

Como todos los módulos en Visual Basic, el de clase tiene propiedades que modifican el comportamiento de la entidad que representa, estas se listan a continuación:

Name	Nombre de la *clase.*
DataBindingBehavior	Establece si es posible enlazar un objeto con un origen de datos.
DataSourceBehavior	Establece si los objetos de esta clase pueden comportarse como un origen de datos para otros objetos.
Persistable	Establece si una clase puede guardar y restaurar datos

entre varios objetos. El valor por omisión es 0 y significa que el objeto no puede ser persistente.

Instancing Establece si es posible o no crear objetos de una clase pública fuera de un proyecto y, en caso de que si sea posible, cómo funcionan. El valor predeterminado es *Private,* que significa que otras aplicaciones no se pueden crear objetos de ella.

Creando una clase

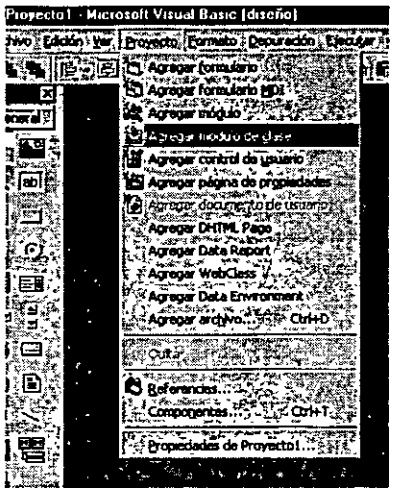
Para crear una clase, hay que realzar los siguientes pasos:

- Crear el módulo que contendrá a la clase y establecer sus propiedades.
- Agregar las propiedades de la clase.
- Agregar los métodos de la clase.
- Agregar los eventos y escribir el código que desencadenen.

Ahora veremos mas detalladamente este proceso:

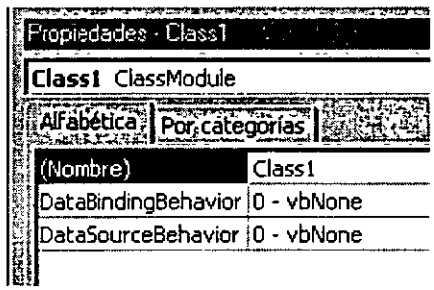
Crear un módulo de clase

Para crear un módulo de clase seleccione *Agregar módulo de clase* del menú *Proyecto* y en entre las opciones resultantes elija *Módulo de clase.* Después asigne al módulo sus propiedades1:



Agregando un modulo de clase.

Siempre que estemos creando una clase estándar, sólo es necesario establecer la propiedad **Name**. Ahora, cuando creemos un servidor *ActiveX*, como veremos en un capítulo posterior, la clase también expondrá su propiedad **Instancing,** que será necesario establecer.



Propiedades del modulo de clase.

Agregar propiedades

La forma más sencilla de definir las propiedades de una clase es agregar variables públicas o privadas al módulo de clase. Por ejemplo:

```
Public peso As Integer
Private conexion As String
```

La propiedad `peso` será accesible desde cualquier parte de la aplicación a través de un objeto de la clase, con la sintaxis: *objeto.propiedad* y la propiedad `conexion` sólo será accesible desde el código del módulo de clase. En este

último caso decimos que la clase oculta la propiedad *conexion.*

Agregar métodos

Los métodos de una clase son procedimientos **Sub** o funciones **Function** escritos en el módulo de clase. De forma predeterminada son declarados públicos. La función de los métodos es proporcionar un acceso controlado a las propiedades de la clase. Por ejemplo:

Métodos

```
Public Function ObtenerPeso() As Integer
   ObtenerPeso = peso  'regresa el peso
End Function

Public Sub AsignarNombre(pes As Integer)
     peso = pes      'establece el valor de peso
End Sub
```

La función *ObtenerPeso* retorna el valor a la propiedad *Nombre,* y la función *AsignarPeso* asigna un valor a la propiedad *Nombre.*

Crear objetos de una clase

Una clase no es una plantilla para crear objetos de esa clase. Para crear un objeto de una clase durante la ejecución, se utiliza la palabra clave **New** de acuerdo con la siguiente sintaxis:

Dim *MiObjeto* **As** New *MiClase*

o

Dim *MiObjeto* **As** *MiClase*

Por ejemplo:

```
Private Cuenta As TextRox     'declara Cuenta de la clase TextBox
Set Cuenta = Texto2           'asigna a Cuenta el objeto Texto2
Cuenta.SetFocus               'fijar la atención en el objeto al que hace
referencia por Cuenta
```

La reutilización de código es la capacidad de utilizar características de un objeto en otro sin la necesidad de volver a escribir su código asociado, como se ha visto antes esto se logra con la herencia. Existen principalmente dos formas de reutilización del código: binario y fuente. La reutilización de código binario se implementa mediante la creación y uso de un objeto en base a su definición original, mientras que la reutilización de código fuente se consigue por herencia, lo que no se permite en Visual Basic.

En una jerarquía de clases, la herencia muestra cómo los objetos que se derivan de otros objetos más simples heredando su comportamiento. Mientras que los modelos de objetos son árboles que describen muestran cómo objetos complejos como un procesador de palabras, contienen colecciones de otros objetos mas simplres, como imágenes. Los controles *ActiveX* normalmente operan como componentes de software reutilizables que se incorporan en cualquier aplicación.

LAS CLASES COMUNES.

Hasta el momento el uso de objetos no nos ha sido del todo extraño, pero tampoco lo ha sido el de las clases, solo que por la estructuración del lenguaje, ha sido invisible para nosotros. Por ejemplo al utilizar controles y Formulario estamos haciendo uso de las plantillas que representan las clases, para generar los objetos correspondientes, solo que en este caso Visual Basic se encarga directamente de ello.

Set *MiObjeto* = **New** *MiClase*

Si utiliza la primera forma, debe recordar que el objeto referenciado por *MiObjeto* de la clase *MiClase* se crea en el momento en que se invoca a una de sus propiedades o métodos; esto es, cuando se ejecute una sentencia de algunas de las formas siguientes:

MiObjeto.propiedad = valor
MiObjeto.método

Es importante recalcar que en Visual Basic existe solo una copia de cada método para ser usada por todos los objetos, no as en el caso de las propiedades, donde existirá una copia por cada uno de ellos

Destruir objetos

La regla principal acerca del tiempo de vida de un objeto es muy sencilla: un objeto se destruye cuando se libera la última referencia al mismo ¿Cómo sabemos acerca de esto? Las reglas de CQM indican que cuando el número de referencias llega a cero, se produce el evento **Terminate** del objeto. Aparte de ésta, no hay más información fiable acerca de este hecho.

Sí es importante saber que cuando una variable sale fuera del ámbito en el que fue definida, o cuando se le asigna el valor **Nothing,** el objeto referenciado por ella se destruye automáticamente, si ésta es la última referencia al mismo.

Interfaz privada y pública

Un aspecto importante del principio de encapsulamiento orientado a objetos es la capacidad de proteger u ocultar los datos o estructura interna de un objeto. Como veremos a continuación, ocultar los datos significa poder efectuar

modificaciones en la implementación de una clase sin afectar al código desarrollado por el usuario de la clase.

¿Cómo se ocultan los datos de un objeto? Hemos visto que esto se hace declarando privadas las propiedades de la clase representativas de esos datos. También podemos declarar privados los métodos que interese de la clase.

Según lo expuesto, la *intefaz privada* está formada por las propiedades y métodos que permanecen ocultos a los usuarios de los objetos de la clase en cuestión; esto es, propiedades y métodos que no son accesibles desde otros módulos, sino que sólo son accesibles desde el módulo de clase que los contiene.

Una interfaz privada permite al diseñador de la clase realizar cambios en su implementación sin que afecte al código escrito por los usuarios de esa clase.

La interfaz pública está formada por las propiedades y métodos a los que los usuarios de los objetos de la clase tienen acceso; esto es, propiedades y métodos accesibles desde otros módulos por estar declarados **Public** a nivel del módulo de clase. Según lo expuesto, hacer que una propiedad sea de sólo lectura es tan sencillo como implementar sólo el procedimiento que da acceso a la obtención del dato.

Agregar propiedades mediante procedimientos

Según lo que hemos visto hasta ahora ¿qué operaciones se pueden realizar con una propiedad? La respuesta es asignarle un valor y obtener su valor.

```
Objeto.asignaPropiedad ( valor)
variable = Objeto.obtenPropiedad
```

Si trasladamos estas operaciones a la propiedad **Caption** de un Formulario, por ejemplo, recordamos que el código utilizado era mucho más intuitivo:

```
Form1.Caption = X
X = Form1.Caption
```

Para poder utilizar las propiedades de un objeto de esta forma hay que definirlas mediante procedimientos **Property** (procedimientos de propiedad). Hay tres clases de procedimientos de propiedad: **Property Get** que retorna el valor de una propiedad, **Property Let** que establece el valor de una propiedad y **Property Set** que establece una referencia a un objeto. Estos procedimientos son generalmente utilizados por parejas; esto es, **Property Get** con Property **Let** y **Property Get** con **Property Set**, y la sintaxis utilizada para invocarlos es:

Property Get: *variable = [objeto.]nombrepropiedad[(argumentos)]*
Property Let: *[objeto.] nombrepropiedad[(argumentos)] = argumento*
Property Set: Set *[objeto.]nombrepropiedad[(argumentos)] = variable*

Es importante recordar que cuando se invoca al procedimiento **Property Let** o al **Property Set,** el argumento especificado a la derecha del signo igual es pasado al último argumento especificado en la definición del mismo. Esto es:

```
PropiedadX(a, b. c) = d

Public Property Let PropiedadX(a, b, c, d)

End Property
```

Esto es, se trata de realizar las operaciones

```
Objeto.asignaPropiedad ( valor)
variable = Objeto.obtenPropiedad
```

de la forma

```
Objeto.Propiedad = valor
variable = Objeto.Propiedad
```

Esta cuestión la resolveremos utilizando los procedimientos **Property Let y Property Get.**

2.0 Fundamentos de la programación de bases de datos.

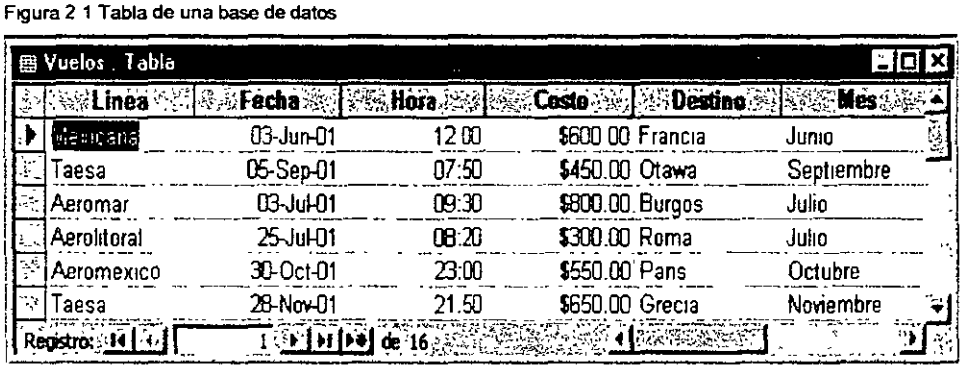
2.1 ¿Qué es una base de datos?

Una base de datos es una colección de datos clasificados y estructurados que son guardados en uno o varios archivos pero referenciados como si de un único archivo se tratara. En la versión 6 de Visual Basic, que es la que nosotros estudiaremos, incluye un Administrador visual de datos que nos permitirá crear y manipular bases de datos *Microsoft Access, Dbase, Foxpro, Paradox, ODBC* y archivos de texto.

Los datos de una base de datos están compuestas por campos y registros. El conjunto de todos los registros forman la base de datos. Una base de datos puede estar formada por una o más *tablas*.

Una tabla es una colección de datos presentada en forma de una matriz bidimensional, donde las filas son los *registros* y las columnas los campos.

Figura 2 1 Tabla de una base de datos



En la figura se puede ver una tabla compuesta por los siguientes campos `Linea, Fecha, Hora, Costo, Destino, Mes.` Así un campo sería Mexicana, y un registro podría ser:

```
Taesa 05-Sep-01 07:50 $600.00 Otawa Septiembre
```

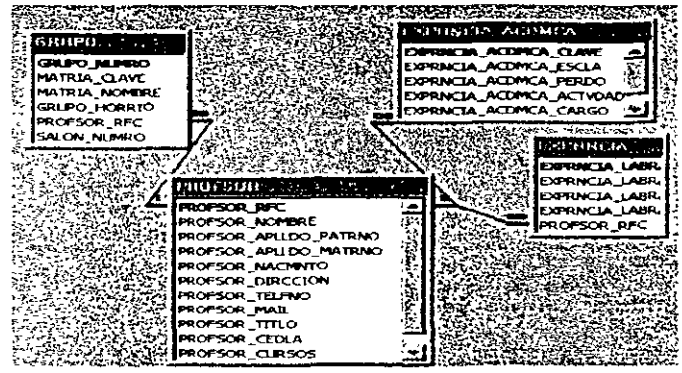
2.2 PROGRAMACIÓN DE BASES DE DATOS

Por lo general en aplicaciones fuertes se vuelve necesario almacenar información, de manera permanente, para esto nos pueden servir los archivos, sin embargo cuando se necesita llevar un orden mas rígido, o bien es necesario hacer consultas de la información almacenada de una manera rápida y segura, no hay nada mejor que una base de datos para almacenar nuestra información.

Las bases de datos actuales, en general, son relacionales. El acceso a

las mismas se hace generalmente utilizando el lenguaje SQL *(Structured Query Language* - Lenguaje de consulta estructurado).

Figura 2.2 Ejemplo de una base de datos relacional



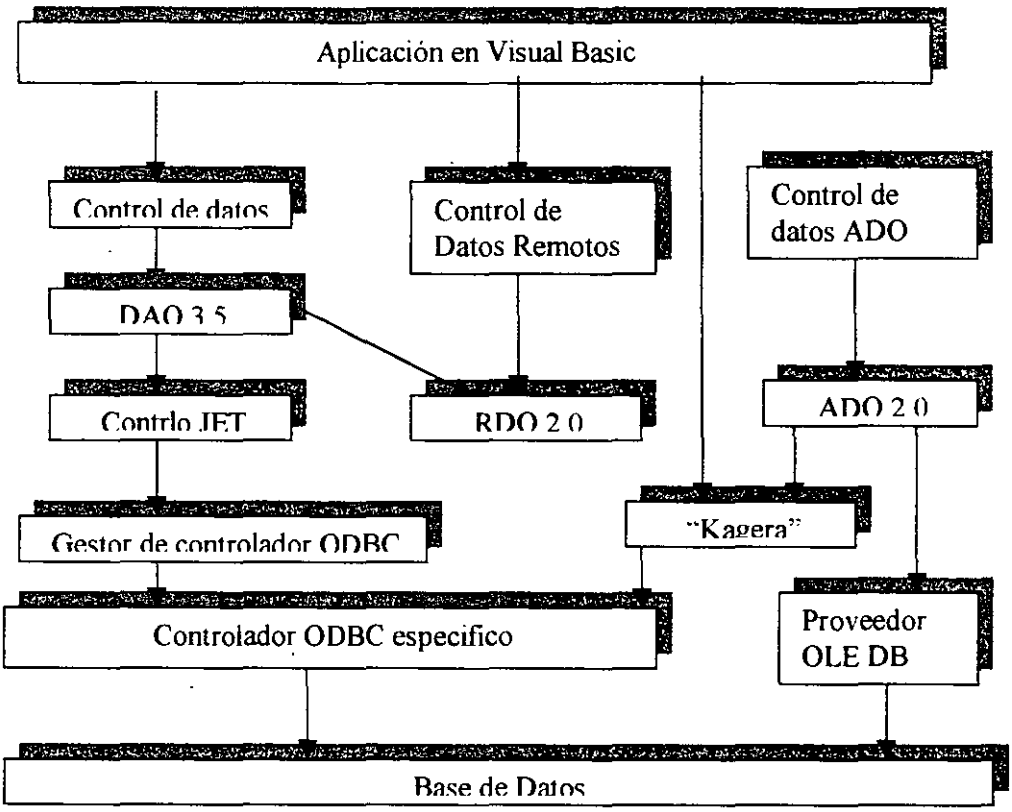
Por otra parte, dada la gran cantidad de situaciones diferentes en las que se puede utilizar una base de datos, es lógico que existan muchos tipos y muchas formas de acceder a ellas. Por eso, para que los programadores puedan acceder de forma estándar a una base de datos, los fabricantes suelen desarrollar junto con la base de datos el controlador ODBC *(Open Database Connectivity* - Conectividad abierta de bases de datos) de acceso correspondiente, que proporciona al programador un conjunto de funciones estándar (API - *Application Programming Inteiftice;* Interfaz de programación de aplicaciones) para acceder al motor de la base.

Algunos otros fabricantes, dotaron a sus herramientas de desarrollo de una nueva capa de software, conocida como motor de bases de datos, intercalada entre el código de la aplicación y el controlador ODBC. Un ejemplo es MS Jet, que mas adelante estudiaremos. El motor de base de datos que proporciona Microsoft en muchos de sus productos (Visual Basic, Visual C++, Excel, Word, etc.). Se trata de una capa de software independiente de la aplicación que lo utiliza para acceder a los datos de la base.

Finalmente, en las herramientas de desarrollo actuales contamos con los controles y los objetos de acceso a datos, los cuales establecen un puente entre la interfaz del usuario y el motor de acceso a datos.

Visual Basic nos proporciona varias formas de acceso a bases de datos remotas, las cuales se describen a continuación:

2.3 Controles para acceso a bases de datos.



Frfgura 2.4 esquema de los diferentes caminos para un enlace a una base de datos.

- Objetos ActiveX *de acceso a datos* (ADO - *ActiveX Data Objects).* Microsoft, a partir de la versión 6 de Visual Basic, introduce el modelo de objetos ADO para el acceso a bases de datos. Este modelo es más sencillo y proporciona mejor integración con las tecnología de Microsoft y con otras tecnologías, una interfaz común para acceso a datos locales y remotos, conjuntos de registros remotos y locales, una interfaz de enlace con los datos accesible para el usuario y un conjuntos de registros jerárquicos.

- Control de datos ADO (ADODC - *ADO Data control).* Permite crear una conexión con una base de datos de una forma fácil y rápida mediante objetos *ActiveX* de acceso a datos.
- Objetos de acceso a datos (DAO - Data Access Objects*).* El modelo de obje tos DAO admite dos entornos diferentes de bases de datos o espacios de tra bajo: *MS Jet* y *acceso directo a ODBC (ODBCDirect).*

Microsoft Jet permite acceder a bases de datos Microsoft *Access,* o bien a bases de datos Microsoft conectadas a ODBC y a orígenes de datos conectados a ISAM *(Indexed Sequential Access Method* - Método de acceso secuencial indexado) para permitir el acceso a formatos de bases de datos externas como son dBASE, Microsoft Excel y Paradox. *MS Jet* carga estos controladores ISAM cuando se hace referencia a ellos en una aplicación.

El *acceso directo a ODBC (ODBCDirect)* permite tener acceso a servidores de bases de datos a través de ODBC sin cargar el motor de base de datos *MS*

- *Control de datos* (DC - *Data Control).* El control *Data* implementa el acceso a los datos mediante el motor de bases de datos *MS Jet,* el mismo motor de bases de datos de *Microsoft Access.*

- *RDO.* Es una interfaz de acceso a datos mediante ODBC orientada a objetos, que incorpora un estilo sencillo de DAO y cuya interfaz expone prácticamente toda la flexibilidad y eficacia de bajo nivel de ODBC. Sin embargo, RDO presenta limitaciones al no proporcionar un acceso apropiado a las bases de datos *MS Jet,* y al sólo permitir el acceso a bases de datos relacionales a través de los controladores ODBC existentes. El control remoto de datos (RDC - *RemoteData Control)* implementa el acceso mediante RDO.

- *Conectividad abierta de bases de datos* (ODBC - Open Database Connectivity*).* Se trata de una interfaz de programación para acceso a servidores de base de datos que proporciona un lenguaje común para las aplicaciones Windows que necesiten acceder a una base de datos en una red. Para utilizar

esta interfaz con distintas bases de datos debe instalar previamente los correspondientes controladores ODBC.

- *VB SQL.* Es una biblioteca de Visual Basic para *SQL Server.* Concretamente se trata de una interfaz de programación (API) para *DB-Library.*

Básicamente las ventajas dependen del tipo de usuario para el que se realice la aplicación.

El motor *Jet* funciona bien en monousuario y en redes pequeñas; tiene sus problemas de bloqueo cuando varios usuarios acceden a una misma información o incluso a informaciones colindantes.

Por esta razón cuando trabajemos con bases de datos grandes es aconsejable trabajar con ODBC. La ventaja de trabajar con el motor *Jet* es que no necesitamos depender de ningún *driver* ODBC externo, que las aplicaciones son más sencillas y dan menos problemas de configuración.

Utilizar RDO es similar en muchos aspectos a utilizar DAO; no obstante hay algunas diferencias, ya que RDO está implementado y diseñado para utilizarlo estrictamente con bases de datos relacionales. RDO no tiene ningún procesador de consultas propio; depende del origen de datos para procesar todas las consultas y crear los conjuntos de resultados. Los objetos de datos propiamente dichos se generan a partir de los conjuntos de resultados y los cursores devueltos por el controlador ODBC.

Finalmente, el acceso a datos basado en OLE DB y ADO es adecuado para una gama amplia de aplicaciones cliente/servidor. Las principales ventajas son fácil utilización, gran velocidad, uso de poca memoria y poca utilización de disco.

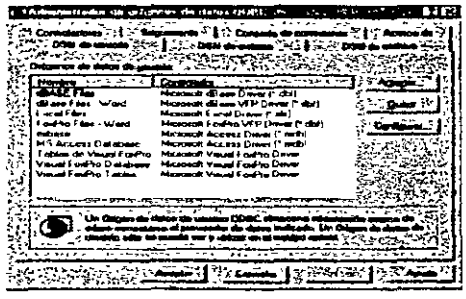
Recordemos, como principio de diseño, que aunque la tecnología posibilite múltiples y enrevesadas posibilidades, las aplicaciones que deben funcionar cada día deben ser lo más sencillas posibles, tanto para el usuario como para el pro gramador que debe soportarlas. Por lo tanto, el modelo de objetos ADO es el que se aconseja utilizar de ahora en adelante.

2.4 Conexión ODBC

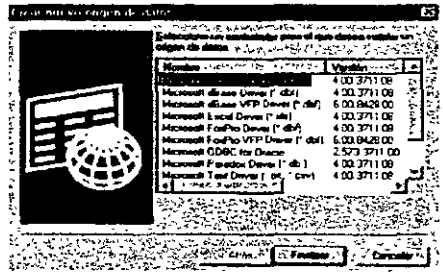
Para poder utilizar una base de datos desde ASP esta debe estar dada de alta para emplearla con controladores ODBC.

Vamos a ver como hacer esta conexión paso a paso.

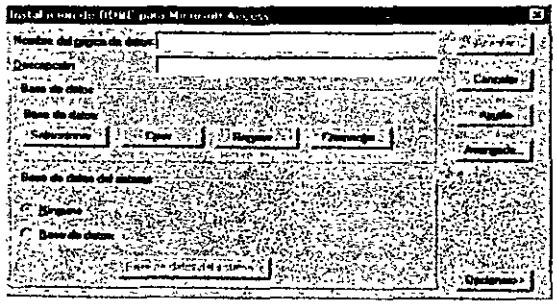
Lo primero será ir a **Inicio -> Configuración -> Panel de Control -> Fuentes ODBC.** Entonces estaremos en una pantalla como la siguiente:



En esta pantalla se encuentran algunos de los controladores para los diferentes tipos de bases de datos que soportan controladores ODBC. Estando ahí damos clic en el botón Agregar..., y se nos mostrará una pantalla como la siguiente:



En ella seleccionamos el tipo de base de datos que tenemos, es decir Acces, FoxPro, Oracle, etc. Y damos clic en el botón Finalizar, entonces estaremos en la siguiente pantalla. En ella tendremos que hacer clic en el botón que dice Seleccionar, en el recuadro que dice Base de Datos. Para que se nos muestre una ventana para seleccionar la base de datos que deseamos agregar.



Después le damos un nombre, en la primera caja de dialogo donde dice: Nombre del origen de datos, con este nombre nosotros haremos la conexión a la base de datos, no tiene que ser el mismo nombre de la base de datos.

Entonces damos clic en Aceptar, y nos regresara a la primera pantalla en ella ahora podremos ver el nombre del origen de datos que le dimos a la conexión acompañado de los controladores para el tipo de base de datos que tenemos. Ahora nuestra base de datos esta lista para ser utilizada desde ASP. Si ahora vamos al explorador de Windows podremos ver que un hay un icono que nos indica que la conexión se realizo satisfactoriamente. En la misma carpeta donde se encuentre la base de datos sobre la que vamos a trabajar encontraremos un icono parecido a esto:

db2.mdb

2.5 Objetos ADO

El modelo de bases de datos basado en Actives Data Objects, está formado por objetos, estos objetos proporcionan una serie de métodos y propiedades con los que podemos acceder fácilmente a las bases de datos .

Para manejar este tipo de bases de datos tenemos siete objetos. De estos siete objetos hay que ver con especial cuidado a tres de ellos que son los mas importantes Connection, Recordset y Command. El resto de los objetos son Field, Parameter, Property y Error, sin embargo para poder emplear estos últimos necesitamos de los primeros.

La descripción de cada uno de los objetos seria:

- Connection: Representa la conexión con una base de datos. Este objeto lo utilizaremos para crear un enlace directo entre nuestra página Web y el servidor de bases de datos. Mientras dure la conexión podremos realizar todas las operaciones que deseemos sobre la base de datos. La conexión terminará cuando nosotros así lo indiquemos con el método Close del mismo objeto.
- RecordSet: representa una tabla de datos. En este objeto serpá donde almacenemos las consultas realizadas a la base de datos a la que

estemos conectados. Estará formada por filas, y por columnas a los que podremos acceder para exponer la información adquirida.

- Command: Representa un comando SQL. Con este objeto podremos ejecutar sentencias SQL sobre la base de datos a la que estemos conectados.

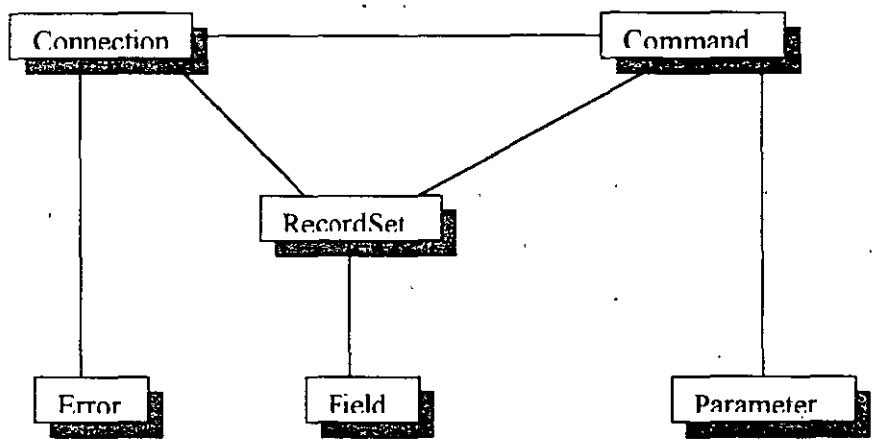
- Field: Representa un campo de un objeto RecordSet. Este objeto solo existirá si existe el correspondiente objeto RecordSet. El objeto RecordSet lleva implícito la colección Fields que representa todos sus campos. Cada elemento de esa colección es un objeto Field.

- Parameter: Representa un parámetro de un procedimiento o cuestión. Nos será de gran ayuda al utilizar el objeto Command para lanzar procedimientos o cuestiones sobre una base de datos.

- Error: Representa un error ADO. Se puede producir un error al realizar la conexión sobre la base de datos. Esto quiere decir que este objeto solo existirá si previamente se ha intentado conectar con una base de datos erróneamente.

- Property: Representa una propiedad específica de un proveedor de datos. Este objeto se encuentra un poco al margen de lso demás, ya que no tiene relación alguna con ellos.

Como se puede suponer entre estos objetos existe cierta relación.



Relaciones:

Connection-RecordSet: Una cualidad que tiene el objeto Connection es la de poder ejeutar comandos SQL. Al hacer esto pueda darse el caso de que el comando devuelva cierto resultado, por ejemplo, si el comando realiza una consulta sobre la base de datos. En tal caso el resultado vendrá dado dentro de un objeto RecordSet.

Command-RecordSet: Al igual que en el caso anterior, al ejecutar una consulta con el objeto Command sobre una base de datos el resultado viene dado en un objeto RecordSet.

Command-Connection: Cuando creamos un objeto Command no será utilizable hasta que no lo relacionamos con una base de datos. Para esto es necesario el objeto Connection. Con este objeto abriremos una sesión con una base de datos y entonces se lo asignaremos al objeto Command como conexión activa. Solo de este modo podremos ejecutar los camandos de este objeto sobre una base de datos.

Connection-Error: El objeto Error solo aparece cuando se produce una conexión errónea a una base de datos. Con este objeto y sus métodos y propiedades podremos determinar la causa del error.

RecordSet-Field: Todo objeto RecordSet tiene entre sus atributos la colección Fields. Esta colección esta formada por objetos del tipo Field. Estos objetos representan ada campo de cada registro del objeto RecordSet. El objeto Field nos permitirá un acceso sencillo a la información del objeto RecordSet.

Command-Parameter: Un comando SQL puede tener parámetros. Esos parámetros pertenecerán al objeto Parameter. Además no sólo podemos ejecutar sentencias simples de SQL con el objteo Command, sino también procedimientos que probablemente llevarán parámetros, tanto de entrada como de salida.

6. Selección de datos mediante SQL

6.1 Introducción

El lenguaje de consulta estructurado (SQL) es un lenguaje de base de datos normalizado, utilizado por el motor de base de datos de Microsoft Jet. SQL se utiliza para crear objetos QueryDef, como el argumento de origen del método OpenRecordSet y como la propiedad RecordSource del control de datos. También se puede utilizar con el método Execute para crear y manipular directamente las bases de datos Jet y crear consultas SQL de paso a través para manipular bases de datos remotas cliente - servidor.

6.2 Componentes del SQL

El lenguaje SQL está compuesto por comandos, cláusulas, operadores y funciones de agregado. Estos elementos se combinan en las instrucciones para crear, actualizar y manipular las bases de datos.

Comandos

Existen dos tipos de comandos SQL:

- Los DLL que permiten crear y definir nuevas bases de datos, campos e índices.
- Los DML que permiten generar consultas para ordenar, filtrar y extraer datos de la base de datos.

Comandos DLL:

Comando	Descripción
CREATE	Utilizado para crear nuevas tablas, campos e índices
DROP	Empleado para eliminar tablas e índices

ALTER	Utilizado para modificar las tablas agregando campos o cambiando la definición de los campos.

Comandos DML:

Comando	Descripción
SELECT	Utilizado para consultar registros de la base de datos que satisfagan un criterio determinado
INSERT	Utilizado para cargar lotes de datos en la base de datos en una única operación.
UPDATE	Utilizado para modificar los valores de los campos y registros especificados
DELETE	Utilizado para eliminar registros de una tabla de una base de datos

Cláusulas

Las cláusulas son condiciones de modificación utilizadas para definir los datos que desea seleccionar o manipular.

Cláusula	Descripción
FROM	Utilizada para especificar la tabla de la cual se van a seleccionar los registros
WHERE	Utilizada para especificar las condiciones que deben reunir los registros que se van a seleccionar
GROUP BY	Utilizada para separar los registros seleccionados en grupos específicos
HAVING	Utilizada para expresar la condición que debe satisfacer cada grupo

ORDER BY	Utilizada para ordenar los registros seleccionados de acuerdo con un orden específico

Operadores lógicos

Operador	Uso
AND	Es el "y" lógico. Evalua dos condiciones y devuelve un valor de verdad sólo si ambas son ciertas.
OR	Es el "o" lógico. Evalúa dos condiciones y devuelve un valor de verdar si alguna de las dos es cierta.
NOT	Negación lógica. Devuelve el valor contrario de la expresión.

Operadores de Comparación

Operador	Uso
<	Menor que
>	Mayor que
<>	Distinto de
<=	Menor ó Igual que
>=	Mayor ó Igual que
=	Igual que
BETWEEN	Utilizado para especificar un intervalo de valores.
LIKE	Utilizado en la comparación de un modelo
In	Utilizado para especificar registros de una base de datos

Funciones de Agregado

Las funciones de agregado se usan dentro de una cláusula SELECT en grupos de registros para devolver un único valor que se aplica a un grupo de registros.

Función	Descripción
AVG	Utilizada para calcular el promedio de los valores de un campo determinado
COUNT	Utilizada para devolver el número de registros de la selección
SUM	Utilizada para devolver la suma de todos los valores de un campo determinado
MAX	Utilizada para devolver el valor más alto de un campo especificado
MIN	Utilizada para devolver el valor más bajo de un campo especificado

Consultas de Selección

Las consultas de selección se utilizan para indicar al motor de datos que devuelva información de las bases de datos, esta información es devuelta en forma de conjunto de registros que se pueden almacenar en un objeto recordset. Este conjunto de registros es modificable.

Consultas Básicas

La sintaxis básica de una consulta de selección es la siguiente:

SELECT Campos FROM Tabla;

En donde campos es la lista de campos que se deseen recuperar y tabla es el origen de los mismos, por ejemplo:

SELECT Nombre, Telefono FROM Clientes;

Esta consulta devuelve un recordset con el campo nombre y teléfono de la tabla clientes.

Ordenar los Registros

Adicionalmente se puede especificar el orden en que se desean recuperar los registros de las tablas mediante la claúsula ORDER BY Lista de Campos. En donde Lista de campos representa los campos a ordenar. Ejemplo:

```
SELECT CodigoPostal, Nombre, Telefono FROM Clientes ORDER
BY Nombre;
```

Esta consulta devuelve los campos CodigoPostal, Nombre, Telefono de la tabla Clientes ordenados por el campo Nombre.

Se pueden ordenar los registros por mas de un campo, como por ejemplo:

```
SELECT CodigoPostal, Nombre, Telefono FROM Clientes ORDER BY
    CodigoPostal, Nombre;
```

Incluso se puede especificar el orden de los registros: ascendente mediante la claúsula ASC (se toma este valor por defecto) ó descendente (DESC).

```
SELECT CodigoPostal, Nombre, Telefono FROM Clientes ORDER BY
    CodigoPostal DESC , Nombre ASC;
```

Consultas con Predicado

El predicado se incluye entre la claúsula y el primer nombre del campo a recuperar, los posibles predicados son:

Predicado	Descripción
ALL	Devuelve todos los campos de la tabla
TOP	Devuelve un determinado número de registros de la tabla
DISTINCT	Omite los registros cuyos campos seleccionados coincidan totalmente
DISTINCTROW	Omite los registros duplicados basandose en la totalidad del registro y no sólo en los campos seleccionados.

ALL

Si no se incluye ninguno de los predicados se asume ALL. El Motor de base de datos selecciona todos los registros que cumplen las condiciones de la instrucción SQL. No se conveniente abusar de este predicado ya que obligamos al motor de la base de datos a analizar la estructura de la tabla para averiguar los campos que contiene, es mucho más rápido indicar el listado de campos deseados. SELECT ALL FROM Empleados; SELECT * FROM Empleados;

TOP

Devuelve un cierto número de registros que entran entre al principio o al final de un rango especificado por una cláusula ORDER BY. Supongamos que queremos recuperar los nombres de los 25 primeros estudiantes del curso 1994:

SELECT TOP 25 Nombre, Apellido FROM Estudiantes ORDER BY Nota DESC;

Si no se incluye la cláusula ORDER BY, la consulta devolverá un conjunto arbitrario de 25 registros de la tabla Estudiantes. El predicado TOP no elige entre valores iguales. En el ejemplo anterior, si la nota media número 25 y la 26 son iguales, la consulta devolverá 26 registros. Se puede utilizar la palabra reservada PERCENT para devolver un cierto porcentaje de registros que caen al principio o al final de un

rango especificado por la cláusula ORDER BY. Supongamos que en lugar de los 25 primeros estudiantes deseamos el 10 por ciento del curso:

```
SELECT TOP 10 PERCENT Nombre, Apellido FROM Estudiantes ORDER
BY Nota DESC;
```

El valor que va a continuación de TOP debe ser un Integer sin signo. TOP no afecta a la posible actualización de la consulta.

DISTINCT

Omite los registros que contienen datos duplicados en los campos seleccionados. Para que los valores de cada campo listado en la instrucción SELECT se incluyan en la consulta deben ser únicos.

Por ejemplo, varios empleados listados en la tabla Empleados pueden tener el mismo apellido. Si dos registros contienen López en el campo Apellido, la siguiente instrucción SQL devuelve un único registro:

```
SELECT DISTINCT Apellido FROM Empleados;
```

Con otras palabras el predicado DISTINCT devuelve aquellos registros cuyos campos indicados en la cláusula SELECT posean un contenido diferente. El resultado de una consulta que utiliza DISTINCT no es actualizable y no refleja los cambios subsiguientes realizados por otros usuarios.

DISTINCTROW

Devuelve los registros diferentes de una tabla; a diferencia del predicado anterior que sólo se fijaba en el contenido de los campos seleccionados, éste lo hace en el contenido del registro completo independientemente de los campo indicados en la cláusula SELECT.

```
SELECT DISTINCTROW Apellido FROM Empleados;
```

Si la tabla empleados contiene dos registros: Antonio López y Marta López, el ejemplo del predicado DISTINCT devuleve un único registro con el valor López en el campo Apellido ya que busca no duplicados en dicho campo. Este último ejemplo devuelve dos registros con el valor López en el apellido ya que se buscan no duplicados en el registro completo.

Alias

En determinadas circunstancias es necesario asignar un nombre a alguna columna determinada de un conjunto devuelto, otras veces por simple capricho o por otras circunstancias. Para resolver todas ellas tenemos la palabra reservada AS que se encarga de asignar el nombre que deseamos a la columna deseada. Tomado como referencia el ejemplo anterior podemos hacer que la columna devuelta por la consulta, en lugar de llamarse apellido (igual que el campo devuelto) se llame Empleado. En este caso procederíamos de la siguiente forma:

```
SELECT DISTINCTROW Apellido AS Empleado FROM Empleados;
```

Bases de Datos Externas

Para concluir este capítulo se debe hacer referencia a la recuperación de registros de bases de datos externa. En ocasiones es necesario la recuperación de información que se encuentra contenida en una tabla que no se encuentra en la base de datos que ejecutará la consulta o que en ese momento no se encuentra abierta, esta situación la podemos salvar con la palabra reservada IN de la siguiente forma:

```
SELECT DISTINCTROW Apellido AS Empleado FROM Empleados
IN 'c:\databases\gestion.mdb';
```

En donde c:\databases\gestion.mdb es la base de datos que contiene la tabla Empleados

Criterios de Selección

En el capítulo anterior se vio la forma de recuperar los registros de las tablas, las formas empleadas devolvían todos los registros de la mencionada tabla. A lo largo de este capítulo se estudiarán las posibilidades de filtrar los registros con el fin de recuperar solamente aquellos que cumplan una condiciones preestablecidas.

Antes de comenzar el desarrollo de este capítulo hay que recalcar tres detalles de vital importancia. El primero de ellos es que cada vez que se desee establecer una condición referida a un campo de texto la condición de búsqueda debe ir encerrada entre comillas simples; la segunda es que no se posible establecer condiciones de búsqueda en los campos memo y; la tercera y última hace referencia a las fechas. Las fechas se deben escribir siempre en formato mm-dd-aa en donde mm representa el mes, dd el día y aa el año, hay que prestar atención a los separadores -no sirve la separación habitual de la barra (/) - hay que utilizar el guión (-) y además la fecha debe ir encerrada entre almohadillas (#). Por ejemplo si deseamos referirnos al día 3 de Septiembre de 1995 deberemos hacerlo de la siguente forma; #09-03-95# ó #9-3-95#.

Operadores Lógicos

Los operadores lógicos soportados por SQL son: AND, OR, XOR, Eqv, Imp, Is y Not. A excepción de los dos últimos todos poseen la siguiente sintaxis:

<expresión1> operador <expresión2>

En donde expresión1 y expresión2 son las condiciones a evaluar, el resultado de la operación varía en función del operador lógico. La tabla adjunta muestra los diferentes posibles resultados:

<expresión1>	Operador	<expresión2>	Resultado
Verdad	AND	Falso	Falso
Verdad	AND	Verdad	Verdad
Falso	AND	Verdad	Falso
Falso	AND	Falso	Falso
Verdad	OR	Falso	Verdad
Verdad	OR	Verdad	Verdad
Falso	OR	Verdad	Verdad
Falso	OR	Falso	Falso
Verdad	XOR	Verdad	Falso
Verdad	XOR	Falso	Verdad
Falso	XOR	Verdad	Verdad
Falso	XOR	Falso	Falso
Verdad	Eqv	Verdad	Verdad
Verdad	Eqv	Falso	Falso
Falso	Eqv	Verdad	Falso
Falso	Eqv	Falso	Verdad
Verdad	Imp	Verdad	Verdad
Verdad	Imp	Falso	Falso
Verdad	Imp	Null	Null
Falso	Imp	Verdad	Verdad
Falso	Imp	Falso	Verdad
Falso	Imp	Null	Verdad
Null	Imp	Verdad	Verdad

Null	Imp	Falso	Null
Null	Imp	Null	Null

Si a cualquiera de las anteriores condiciones le anteponemos el operador NOT el resultado de la operación será el contrario al devuelto sin el operador NOT.

El último operador denominado IS se emplea para comparar dos variables de tipo objeto <Objeto1> Is <Objeto2>. Este operador devuelve verdad si los dos objetos son iguales.

```
SELECT * FROM Empleados WHERE Edad > 25 AND Edad < 50;
SELECT * FROM Empleados WHERE (Edad > 25 AND Edad < 50) OR Sueldo
= 100;
SELECT * FROM Empleados WHERE NOT Estado = 'Soltero';
SELECT * FROM Empleados WHERE (Sueldo > 100 AND Sueldo < 500) OR
        (Provincia = 'Madrid' AND Estado = 'Casado');
```

Intervalos de Valores

Para indicar que deseamos recuperar los registros según el intervalo de valores de un campo emplearemos el operador Between cuya sintaxis es:

campo [Not] Between valor1 And valor2 (la condición Not es opcional)

En este caso la consulta devolvería los registros que contengan en "campo" un valor incluido en el intervalo valor1, valor2 (ambos inclusive). Si anteponemos la condición Not devolverá aquellos valores no incluidos en el intervalo.

```
SELECT * FROM Pedidos WHERE CodPostal Between 28000 And 28999;
(Devuelve los pedidos realizados en la provincia de Madrid)
SELECT IIf(CodPostal Between 28000 And 28999, 'Provincial', 'Nacional')
FROM Editores;
(Devuelve el valor 'Provincial' si el código postal se encuentra en el intervalo,
```

'Nacional' en caso contrario)

El Operador Like

Se utiliza para comparar una expresión de cadena con un modelo en una expresión SQL. Su sintaxis es:

expresión Like modelo

En donde expresión es una cadena modelo o campo contra el que se compara expresión. Se puede utilizar el operador Like para encontrar valores en los campos que coincidan con el modelo especificado. Por modelo puede especificar un valor completo (Ana María), o se pueden utilizar caracteres comodín como los reconocidos por el sistema operativo para encontrar un rango de valores (Like An*).

El operador Like se puede utilizar en una expresión para comparar un valor de un campo con una expresión de cadena. Por ejemplo, si introduce Like C* en una consulta SQL, la consulta devuelve todos los valores de campo que comiencen por la letra C. En una consulta con parámetros, puede hacer que el usuario escriba el modelo que se va a utilizar.

El ejemplo siguiente devuelve los datos que comienzan con la letra P seguido de cualquier letra entre A y F y de tres dígitos:

Like 'P[A-F]###'

Este ejemplo devuelve los campos cuyo contenido empiece con una letra de la A a la D seguidas de cualquier cadena.

Like '[A-D]*'

En la tabla siguiente se muestra cómo utilizar el operador Like para comprobar expresiones con diferentes modelos.

Tipo de coincidencia	Modelo Planteado	Coincide	No coincide
Varios caracteres	'a*a'	'aa', 'aBa', 'aBBBa'	'aBC'
Carácter especial	'a[*]a'	'a*a'	'aaa'
Varios caracteres	'ab*'	'abcdefg', 'abc'	'cab', 'aab'
Un solo carácter	'a?a'	'aaa', 'a3a', 'aBa'	'aBBBa'
Un solo dígito	'a#a'	'a0a', 'a1a', 'a2a'	'aaa', 'a10a'
Rango de caracteres	'[a-z]'	'f', 'p', 'j'	'2', '&'
Fuera de un rango	'[!a-z]'	'9', '&', '%'	'b', 'a'
Distinto de un dígito	'[!0-9]'	'A', 'a', '&', '~'	'0', '1', '9'
Combinada	'a[!b-m]#'	'An9', 'az0', 'a99'	'abc', 'aj0'

El Operador In

Este operador devuelve aquellos registros cuyo campo indicado coincide con alguno de los en una lista. Su sintaxis es:

expresión [Not] In(valor1, valor2, . . .)

SELECT * FROM Pedidos WHERE Provincia In ('Madrid', 'Barcelona', 'Sevilla');

La cláusula WHERE

La cláusula WHERE puede usarse para determinar qué registros de las tablas enumeradas en la cláusula FROM aparecerán en los resultados de la instrucción SELECT. Depués de escribir esta cláusula se deben especificar las condiciones expuestas en los dos primeros apartados de este capítulo. Si no se emplea esta cláusula, la consulta devolverá todas las filas de la tabla. WHERE es opcional, pero cuando aparece debe ir a continuación de FROM.

```
SELECT Apellidos, Salario FROM Empleados WHERE Salario > 21000;
    SELECT Id_Producto, Existencias FROM Productos
    WHERE Existencias <= Nuevo_Pedido;

SELECT * FROM Pedidos WHERE Fecha_Envio = #5/10/94#;
SELECT Apellidos, Nombre FROM Empleados WHERE Apellidos = 'King';

SELECT Apellidos, Nombre FROM Empleados WHERE Apellidos Like 'S*';
SELECT Apellidos, Salario FROM Empleados WHERE Salario Between
200 And 300;

SELECT Apellidos, Salario FROM Empl WHERE Apellidos Between 'Lon' And
'Tol';
SELECT Id_Pedido, Fecha_Pedido FROM Pedidos WHERE Fecha_Pedido
    Between #1-1-94# And  #30-6-94#;

SELECT Apellidos, Nombre, Ciudad FROM Empleados WHERE Ciudad
    In ('Sevilla', 'Los Angeles', 'Barcelona');
```

Agrupamiento de Registros y Funciones Agregadas

La cláusula GROUP BY

Combina los registros con valores idénticos, en la lista de campos especificados, en un único registro. Para cada registro se crea un valor sumario si se incluye una función SQL agregada, como por ejemplo Sum o Count, en la instrucción SELECT. Su sintaxis es:

SELECT campos FROM tabla WHERE criterio GROUP BY campos del grupoGROUP BY es opcional. Los valores de resumen se omiten si no existe una función SQL agregada en la instrucción SELECT. Los valores Null en los campos GROUP BY se agrupan y no se omiten. No obstante, los valores Null no se evalúan en ninguna de las funciones SQL agregadas.

Se utiliza la cláusula WHERE para excluir aquellas filas que no desea agrupar, y la cláusula HAVING para filtrar los registros una vez agrupados.

A menos que contenga un dato Memo u Objeto OLE , un campo de la lista de campos GROUP BY puede referirse a cualquier campo de las tablas que aparecen en la cláusula FROM, incluso si el campo no esta incluido en la instrucción SELECT, siempre y cuando la instrucción SELECT incluya al menos una función SQL agregada.

Todos los campos de la lista de campos de SELECT deben o bien incluirse en la cláusula GROUP BY o como argumentos de una función SQL agregada.

```
SELECT Id_Familia, Sum(Stock) FROM Productos GROUP BY Id_Familia;
```

Una vez que GROUP BY ha combinado los registros, HAVING muestra cualquier registro agrupado por la cláusula GROUP BY que satisfaga las condiciones de la cláusula HAVING.

HAVING es similar a WHERE, determina qué registros se seleccionan. Una vez que los registros se han agrupado utilizando GROUP BY, HAVING determina cuales de ellos se van a mostrar.

```
SELECT Id_Familia Sum(Stock) FROM Productos GROUP BY Id_Familia
        HAVING Sum(Stock) > 100 AND NombreProducto Like BOS*;
```

AVG (Media Aritmética)

Calcula la media aritmética de un conjunto de valores contenidos en un campo especificado de una consulta. Su sintaxis es la siguiente:

```
Avg(expr)
```

En donde expr representa el campo que contiene los datos numéricos para los que se desea calcular la media o una expresión que realiza un cálculo utilizando los datos de dicho campo. La media calculada por Avg es la media aritmética (la suma de los valores dividido por el número de valores). La función Avg no incluye ningún campo Null en el cálculo.

```
SELECT Avg(Gastos) AS Promedio FROM Pedidos WHERE Gastos > 100;
```

Count (Contar Registros)

Calcula el número de registros devueltos por una consulta. Su sintaxis es la siguiente:

```
Count(expr)
```

En donde expr contiene el nombre del campo que desea contar. Los operandos de expr pueden incluir el nombre de un campo de una tabla, una constante o una función (la cual puede ser intrínseca o definida por el usuario pero no otras de las funciones agregadas de SQL). Puede contar cualquier tipo de datos incluso texto.

Aunque expr puede realizar un cálculo sobre un campo, Count simplemente cuenta el número de registros sin tener en cuenta qué valores se almacenan en los registros. La función Count no cuenta los registros que tienen campos null a menos que expr sea el carácter comodín asterisco (*). Si utiliza un asterisco, Count calcula el número total de registros, incluyendo aquellos que contienen campos null. Count(*) es considerablemente más rápida que Count(Campo). No se debe poner el asterisco entre dobles comillas ('*').

```
SELECT Count(*) AS Total  FROM Pedidos;
```

Si expr identifica a múltiples campos, la función Count cuenta un registro sólo si al menos uno de los campos no es Null. Si todos los campos especificados son Null, no se cuenta el registro. Hay que separar los nombres de los campos con ampersand (&).

```
SELECT Count(FechaEnvío & Transporte) AS Total FROM Pedidos;
```

Max y Min (Valores Máximos y Mínimos)

Devuelven el mínimo o el máximo de un conjunto de valores contenidos en un campo especifico de una consulta. Su sintaxis es:

```
Min(expr)
Max(expr)
```

En donde expr es el campo sobre el que se desea realizar el cálculo. Expr pueden incluir el nombre de un campo de una tabla, una constante o una función (la cual puede ser intrínseca o definida por el usuario pero no otras de las funciones agregadas de SQL).

```
SELECT Min(Gastos) AS ElMin FROM Pedidos WHERE Pais = 'España';
SELECT Max(Gastos) AS ElMax FROM Pedidos WHERE Pais  = 'España';
```

StDev y StDevP (Desviación Estándar)

Devuelve estimaciones de la desviación estándar para la población (el total de los registros de la tabla) o una muestra de la población representada (muestra aleatoria) . Su sintaxis es:

StDev(expr)
StDevP(expr)

En donde expr representa el nombre del campo que contiene los datos que desean evaluarse o una expresión que realiza un cálculo utilizando los datos de dichos campos. Los operandos de expr pueden incluir el nombre de un campo de una tabla, una constante o una función (la cual puede ser intrínseca o definida por el usuario pero no otras de las funciones agregadas de SQL).

StDevP evalúa una población, y StDev evalúa una muestra de la población. Si la consulta contiene menos de dos registros (o ningún registro para StDevP), estas funciones devuelven un valor Null (el cual indica que la desviación estándar no puede calcularse).

SELECT StDev(Gastos) AS Desviacion FROM Pedidos WHERE Pais = 'España';
SELECT StDevP(Gastos) AS Desviacion FROM Pedidos WHERE Pais= 'España';

Sum (Sumar Valores)

Devuelve la suma del conjunto de valores contenido en un campo especifico de una consulta. Su sintaxis es:

Sum(expr)

En donde expr respresenta el nombre del campo que contiene los datos que desean sumarse o una expresión que realiza un cálculo utilizando los datos de dichos campos. Los operandos de expr pueden incluir el nombre de un campo de una tabla,

Consultas de Eliminación

DELETE crea una consulta de eliminación que elimina los registros de una o más de las tablas listadas en la cláusula FROM que satisfagan la cláusula WHERE. Esta consulta elimina los registros completos, no es posible eliminar el contenido de algún campo en concreto. Su sintaxis es:

DELETE Tabla.* FROM Tabla WHERE criterio

DELETE es especialmente útil cuando se desea eliminar varios registros. En una instrucción DELETE con múltiples tablas, debe incluir el nombre de tabla (Tabla.*). Si especifica más de una tabla desde la que eliminar registros, todas deben ser tablas de muchos a uno. Si desea eliminar todos los registros de una tabla, eliminar la propia tabla es más eficiente que ejecutar una consulta de borrado.

Se puede utilizar DELETE para eliminar registros de una única tabla o desde varios lados de una relación uno a muchos. Las operaciones de eliminación en cascada en una consulta únicamente eliminan desde varios lados de una relación. Por ejemplo, en la relación entre las tablas Clientes y Pedidos, la tabla Pedidos es la parte de muchos por lo que las operaciones en cascada solo afectaran a la tabla Pedidos. Una consulta de borrado elimina los registros completos, no únicamente los datos en campos específicos. Si desea eliminar valores en un campo especificado, crear una consulta de actualización que cambie los valores a Null.

Una vez que se han eliminado los registros utilizando una consulta de borrado, no puede deshacer la operación. Si desea saber qué registros se eliminarán, primero examine los resultados de una consulta de selección que utilice el mismo criterio y después ejecute la consulta de borrado. Mantenga copias de seguridad de sus datos en todo momento. Si elimina los registros equivocados podrá recuperarlos desde las copias de seguridad.

DELETE * FROM Empleados WHERE Cargo = 'Vendedor';

Consultas de Datos Añadidos

una constante o una función (la cual puede ser intrínseca o definida por el usuario pero no otras de las funciones agregadas de SQL).

SELECT Sum(PrecioUnidad * Cantidad) AS Total FROM DetallePedido;

Var y VarP (Varianza)

Devuelve una estimación de la varianza de una población (sobre el total de los registros) o una muestra de la población (muestra aleatoria de registros) sobre los valores de un campo. Su sintaxis es:

Var(expr)
VarP(expr)

VarP evalúa una población, y Var evalúa una muestra de la población. Expr el nombre del campo que contiene los datos que desean evaluarse o una expresión que realiza un cálculo utilizando los datos de dichos campos. Los operandos de expr pueden incluir el nombre de un campo de una tabla, una constante o una función (la cual puede ser intrínseca o definida por el usuario pero no otras de las funciones agregadas de SQL).

Si la consulta contiene menos de dos registros, Var y VarP devuelven Null (esto indica que la varianza no puede calcularse). Puede utilizar Var y VarP en una expresión de consulta o en una Instrucción SQL.

SELECT Var(Gastos) AS Varianza FROM Pedidos WHERE Pais = 'España';
SELECT VarP(Gastos) AS Varianza FROM Pedidos WHERE Pais = 'España';

Consultas de Acción

Las consultas de acción son aquellas que no devuelven ningún registro, son las encargadas de acciones como añadir y borrar y modificar registros.

INSERT INTO agrega un registro en una tabla. Se la conoce como una consulta de datos añadidos. Esta consulta puede ser de dos tipos: Insertar un único registro ó Insertar en una tabla los registros contenidos en otra tabla.

Insertar un único Registro

En este caso la sintaxis es la siguiente:

```
INSERT INTO Tabla (campo1, campo2, .., campoN)
VALUES (valor1, valor2, ..., valorN)
```

Esta consulta graba en el campo1 el valor1, en el campo2 y valor2 y así sucesivamente. Hay que prestar especial atención a acotar entre comillas simples (') los valores literales (cadenas de caracteres) y las fechas indicarlas en formato mm-dd-aa y entre caracteres de almohadillas (#).

Insertar Registros de otra Tabla

En este caso la sintaxis es:

```
INSERT INTO Tabla [IN base_externa] (campo1, campo2, ..., campoN)
SELECT TablaOrigen.campo1, TablaOrigen.campo2, ..., TablaOrigen.campoN
FROM TablaOrigen
```

En este caso se seleccionarán los campos 1,2, ..., n de la tabla origen y se grabarán en los campos 1,2,.., n de la Tabla. La condición SELECT puede incluir la cláusula WHERE para filtrar los registros a copiar. Si Tabla y TablaOrigen poseen la misma estrucutra podemos simplificar la sintaxis a:

```
INSERT INTO Tabla  SELECT TablaOrigen.* FROM TablaOrigen
```

De esta forma los campos de TablaOrigen se grabarán en Tabla, para realizar esta operación es necesario que todos los campos de TablaOrigen estén contenidos

con igual nombre en Tabla. Con otras palabras que Tabla posea todos los campos de TablaOrigen (igual nombre e igual tipo).

En este tipo de consulta hay que tener especial atención con los campos contadores o autonuméricos puesto que al insertar un valor en un campo de este tipo se escribe el valor que contenga su campo homólogo en la tabla origen, no incrementandose como le corresponde.

Se puede utilizar la instrucción INSERT INTO para agregar un registro único a una tabla, utilizando la sintaxis de la consulta de adición de registro único tal y como se mostró anteriormente. En este caso, su código especifica el nombre y el valor de cada campo del registro. Debe especificar cada uno de los campos del registro al que se le va a asignar un valor así como el valor para dicho campo. Cuando no se especifica dicho campo, se inserta el valor predeterminado o Null. Los registros se agregan al final de la tabla.

También se puede utilizar INSERT INTO para agregar un conjunto de registros pertenecientes a otra tabla o consulta utilizando la cláusula SELECT ... FROM como se mostró anteriormente en la sintaxis de la consulta de adición de múltiples registros. En este caso la cláusula SELECT especifica los campos que se van a agregar en la tabla destino especificada.

La tabla destino u origen puede especificar una tabla o una consulta.

Si la tabla destino contiene una clave principal, hay que segurarse que es única, y con valores no-Null ; si no es así, no se agregarán los registros. Si se agregan registros a una tabla con un campo Contador , no se debe incluir el campo Contador en la consulta. Se puede emplear la cláusula IN para agregar registros a una tabla en otra base de datos.

Se pueden averiguar los registros que se agregarán en la consulta ejecutando primero una consulta de selección que utilice el mismo criterio de selección y ver el resultado. Una consulta de adición copia los registros de una o más tablas en otra. Las

tablas que contienen los registros que se van a agregar no se verán afectadas por la consulta de adición. En lugar de agregar registros existentes en otra tabla, se puede especificar los valores de cada campo en un nuevo registro utilizando la cláusula VALUES. Si se omite la lista de campos, la cláusula VALUES debe incluir un valor para cada campo de la tabla, de otra forma fallará INSERT.

```
INSERT INTO Clientes SELECT Clientes_Viejos.* FROM Clientes_Nuevos;
INSERT INTO Empleados (Nombre, Apellido, Cargo)
VALUES ('Luis', 'Sánchez', 'Becario');

INSERT INTO Empleados SELECT Vendedores.* FROM Vendedores
WHERE Fecha_Contratacion < Now() - 30;
```

Consultas de Actualización

UPDATE

crea una consulta de actualización que cambia los valores de los campos de una tabla especificada basándose en un criterio específico. Su sintaxis es:

```
UPDATE Tabla SET Campo1=Valor1, Campo2=Valor2, ... CampoN=ValorN
WHERE Criterio;
```

UPDATE es especialmente útil cuando se desea cambiar un gran número de registros o cuando éstos se encuentran en múltiples tablas. Puede cambiar varios campos a la vez. El ejemplo siguiente incrementa los valores Cantidad pedidos en un 10 por ciento y los valores Transporte en un 3 por ciento para aquellos que se hayan enviado al Reino Unido:

```
UPDATE Pedidos SET Pedido = Pedidos * 1.1, Transporte = Transporte *
1.03
WHERE PaisEnvío = 'ES';
```

UPDATE

no genera ningún resultado. Para saber qué registros se van a cambiar, hay que examinar primero el resultado de una consulta de selección que utilice el mismo criterio y después ejecutar la consulta de actualización.

UPDATE Empleados SET Grado = 5 WHERE Grado = 2;

UPDATE Productos SET Precio = Precio * 1.1 WHERE Proveedor = 8 AND Familia = 3;

Si en una consulta de actualización suprimimos la cláusula

WHERE

todos los registros de la tabla señalada serán actualizados.

UPDATE Empleados SET Salario = Salario * 1.1

Tipos de datos

Los tipos de datos SQL se clasifican en 13 tipos de datos primarios y de varios sinónimos válidos reconocidos por dichos tipos de datos.

Tipos de datos primarios:

Tipo de **os**	**Longitud**	**Descripción**
BINARY	1 byte	Para consultas sobre tabla adjunta de productos ases de datos que definen un tipo de datos Binario.
BIT	1 byte	Valores Si/No ó True/False
BYTE	1 byte	Un valor entero entre 0 y 255.
COUNTER	4 bytes	Un número incrementado automáticamente (de Long)
CURRENCY	8 bytes	Un entero escalable entre 337.203.685.477,5808 y 922.337.203.685.477,5807.
DATETIME	8 bytes	Un valor de fecha u hora entre los años 100 y

SINGLE	4 bytes	Un valor en punto flotante de precisión simple con rango de -3.402823*1038a -1.401298*10-45para res negativos, 1.401298*10-45a 3.402823*1038para res positivos, y 0.
DOUBLE	8 bytes	Un valor en punto flotante de doble precisión con rango de -1.79769313486232*10308a -065645841247*10-324para valores negativos, 065645841247*10-324a 769313486232*10308para valores positivos, y 0.
SHORT	2 bytes	Un entero corto entre -32,768 y 32,767.
LONG	4 bytes	Un entero largo entre -2,147,483,648 y 7,483,647.
LONGTEXT	1 byte por cter	De cero a un máximo de 1.2 gigabytes.
LONGBINARY	Según se site	De cero 1 gigabyte. Utilizado para objetos OLE.
TEXT	1 byte por cter	De cero a 255 caracteres.

La siguiente tabla recoge los sinonimos de los tipos de datos definidos:

Tipo de Dato	**Sinónimos**
BINARY	VARBINARY
BIT	BOOLEAN ICAL ICAL1 NO
BYTE	INTEGER1

COUNTER	AUTOINCREMENT
CURRENCY	MONEY
DATETIME	DATE E ESTAMP
SINGLE	FLOAT4 ISINGLE L
DOUBLE	FLOAT AT8 DOUBLE IBER IERIC
SHORT	INTEGER2 LLINT
LONG	INT GER GER4
LONGBINARY	GENERAL OBJECT
LONGTEXT	LONGCHAR IO E
TEXT	ALPHANUMERIC R RACTER NG CHAR

VARIANT (No Admitido)	VALUE

Subconsultas

Una subconsulta es una instrucción SELECT anidada dentro de una instrucción SELECT, SELECT...INTO, INSERT...INTO, DELETE, o UPDATE o dentro de otra subconsulta.

Puede utilizar tres formas de sintaxis para crear una subconsulta:

comparación [ANY | ALL | SOME] (instrucción sql)
expresión [NOT] IN (instrucción sql)
[NOT] EXISTS (instrucción sql)

En donde:

comparación

Es una expresión y un operador de comparación que compara la expresión con el resultado de la subconsulta.

expresión

Es una expresión por la que se busca el conjunto resultante de la subconsulta.

instrucción sql

Es una instrucción SELECT, que sigue el mismo formato y reglas que cualquier otra instrucción SELECT. Debe ir entre paréntesis.

Se puede utilizar una subconsulta en lugar de una expresión en la lista de campos de una instrucción SELECT o en una cláusula WHERE o HAVING. En una subconsulta, se utiliza una instrucción SELECT para proporcionar un conjunto de uno o más valores especificados para evaluar en la expresión de la cláusula WHERE o HAVING.

Se puede utilizar el predicado ANY o SOME, los cuales son sinónimos, para recuperar registros de la consulta principal, que satisfagan la comparación con cualquier otro registro recuperado en la subconsulta. El ejemplo siguiente devuelve todos los productos cuyo precio unitario es mayor que el de cualquier producto vendido con un descuento igual o mayor al 25 por ciento:

```
SELECT * FROM Productos WHERE PrecioUnidad > ANY
(SELECT PrecioUnidad FROM DetallePedido WHERE Descuento >= 0 .25);
```

El predicado ALL se utiliza para recuperar únicamente aquellos registros de la consulta principal que satisfacen la comparación con todos los registros recuperados en la subconsulta. Si se cambia ANY por ALL en el ejemplo anterior, la consulta devolverá únicamente aquellos productos cuyo precio unitario sea mayor que el de todos los productos vendidos con un descuento igual o mayor al 25 por ciento. Esto es mucho más restrictivo.

El predicado IN se emplea para recuperar únicamente aquellos registros de la consulta principal para los que algunos registros de la subconsulta contienen un valor igual. El ejemplo siguiente devuelve todos los productos vendidos con un descuento igual o mayor al 25 por ciento:

```
SELECT * FROM Productos WHERE IDProducto IN
(SELECT IDProducto FROM DetallePedido WHERE Descuento >=  0.25);
```

Inversamente se puede utilizar NOT IN para recuperar únicamente aquellos registros de la consulta principal para los que no hay ningún registro de la subconsulta que contenga un valor igual.

El predicado EXISTS (con la palabra reservada NOT opcional) se utiliza en comparaciones de verdad/falso para determinar si la subconsulta devuelve algún registro.

Se puede utilizar también alias del nombre de la tabla en una subconsulta para referirse a tablas listadas en la cláusula FROM fuera de la subconsulta. El ejemplo

siguiente devuelve los nombres de los empleados cuyo salario es igual o mayor que el salario medio de todos los empleados con el mismo título. A la tabla Empleados se le ha dado el alias T1:

```
SELECT Apellido, Nombre, Titulo, Salario FROM Empleados AS T1
WHERE Salario >= (SELECT Avg(Salario) FROM Empleados
WHERE T1.Titulo = Empleados.Titulo) ORDER BY Titulo;
```

En el ejemplo anterior , la palabra reservada AS es opcional. Otros ejemplos:

```
SELECT Apellidos, Nombre, Cargo, Salario FROM Empleados
WHERE Cargo LIKE "Agente Ven*" AND Salario > ALL (SELECT Salario
FROM
Empleados WHERE (Cargo LIKE "*Jefe*") OR (Cargo LIKE "*Director*"));
```

Obtiene una lista con el nombre, cargo y salario de todos los agentes de ventas cuyo salario es mayor que el de todos los jefes y directores.

```
SELECT DISTINCTROW NombreProducto, Precio_Unidad FROM Productos
WHERE (Precio_Unidad = (SELECT Precio_Unidad FROM Productos
WHERE
Nombre_Producto = "Almíbar anisado");
```

Obtiene una lista con el nombre y el precio unitario de todos los productos con el mismo precio que el almíbar anisado.

```
SELECT DISTINCTROW Nombre_Contacto, Nombre_Compañia,
Cargo_Contacto,
Telefono FROM Clientes WHERE (ID_Cliente IN (SELECT DISTINCTROW
ID_Cliente FROM Pedidos WHERE Fecha_Pedido >= #04/1/93#
<#07/1/93#);
```

Obtiene una lista de las compañías y los contactos de todos los clientes que han realizado un pedido en el segundo trimestre de 1993.

```
SELECT Nombre, Apellidos FROM Empleados AS E WHERE EXISTS
(SELECT * FROM Pedidos AS O WHERE O.ID_Empleado =
E.ID_Empleado);
```

Selecciona el nombre de todos los empleados que han reservado al menos un pedido.

SELECT DISTINCTROW Pedidos.Id_Producto, Pedidos.Cantidad,
(SELECT DISTINCTROW Productos.Nombre FROM Productos WHERE
Productos.Id_Producto = Pedidos.Id_Producto) AS ElProducto FROM
Pedidos WHERE Pedidos.Cantidad > 150 ORDER BY Pedidos.Id_Producto;

Recupera el Código del Producto y la Cantidad pedida de la tabla pedidos, extrayendo el nombre del producto de la tabla de productos.

Consultas de Referencias Cruzadas

Una consulta de referencias cruzadas es aquella que nos permite visualizar los datos en filas y en columnas, estilo tabla, por ejemplo:

Producto / Año	1996	1997
Pantalones	1.250	3.000
Camisas	8.560	1.253
Zapatos	4.369	2.563

Si tenemos una tabla de productos y otra tabla de pedidos, podemos visualizar en total de productos pedidos por año para un artículo determinado, tal y como se visualiza en la tabla anterior.

La sintaxis para este tipo de consulta es la siguiente:

TRANSFORM función agregada instrucción select PIVOT campo pivot
[IN (valor1[, valor2[, ...]])]

En donde:

función agregada

Es una función SQL agregada que opera sobre los datos seleccionados.

instrucción select

Es una instrucción SELECT.

campo pivot

Es el campo o expresión que desea utilizar para crear las cabeceras de la columna en el resultado de la consulta.

valor1, valor2

Son valores fijos utilizados para crear las cabeceras de la columna.

Para resumir datos utilizando una consulta de referencia cruzada, se seleccionan los valores de los campos o expresiones especificadas como cabeceras de columnas de tal forma que pueden verse los datos en un formato más compacto que con una consulta de selección.

TRANSFORM es opcional pero si se incluye es la primera instrucción de una cadena SQL. Precede a la instrucción SELECT que especifica los campos utilizados como encabezados de fila y una cláusula GROUP BY que especifica el agrupamiento de las filas. Opcionalmente puede incluir otras cláusulas como por ejemplo WHERE, que especifica una selección adicional o un criterio de ordenación.

Los valores devueltos en campo pivot se utilizan como encabezados de columna en el resultado de la consulta. Por ejemplo, al utilizar las cifras de ventas en el mes de la venta como pivot en una consulta de referencia cruzada se crearían 12 columnas. Puede restringir el campo pivot para crear encabezados a partir de los valores fijos (valor1, valor2) listados en la cláusula opcional IN.

También puede incluir valores fijos, para los que no existen datos, para crear columnas adicionales.

Ejemplos

TRANSFORM Sum(Cantidad) AS Ventas SELECT Producto, Cantidad FROM

Pedidos WHERE Fecha Between #01-01-98# And #12-31-98# GROUP BY Producto

ORDER BY Producto PIVOT DatePart("m", Fecha);

Crea una consulta de tabla de referencias cruzadas que muestra lasn ventas de productos por mes para un año específico. Los meses aparecen de izquierda a derecha como columnas y los nombres de los productos aparecen de arriba hacia abajo como filas.

TRANSFORM Sum(Cantidad) AS Ventas SELECT Compania FROM Pedidos

WHERE Fecha Between #01-01-98# And #12-31-98# GROUP BY Compania

ORDER BY Compania PIVOT "Trimestre " & DatePart("q", Fecha) In ('Trimestre1',

'Trimestre2', 'Trimestre 3', 'Trimestre 4');

Crea una consulta de tabla de referencias cruzadas que muestra las ventas de productos por trimestre de cada proveedor en el año indicado. Los trimestres aparecen de izquierda a derecha como columnas y los nombres de los proveedores aparecen de arriba hacia abajo como filas.

Un caso práctico:

Se trata de resolver el siguiente problema: tenemos una tabla de productos con dos campos, el código y el nombre del producto, tenemos otra tabla de pedidos en la que anotamos el código del producto, la fecha del pedido y la cantidad pedida. Deseamos consultar los totales de producto por año, calculando la media anual de ventas.

Estructura y datos de las tablas:

1. Artículos:

ID	Nombre

1	Zapatos
2	Pantalones
3	Blusas

2. Pedidos:

Id	Fecha	Cantidad
1	11/11/1996	250
2	11/11/1996	125
3	11/11/1996	520
1	12/10/1996	50
2	04/05/1996	250
3	05/08/1996	100
1	01/01/1997	40
2	02/08/1997	60
3	05/10/1997	70
1	12/12/1997	8
2	15/12/1997	520
3	17/10/1997	1250

Para resolver la consulta planteamos la siguiente consulta:

TRANSFORM Sum(Pedidos.Cantidad) AS Resultado SELECT Nombre AS Producto,

Pedidos.Id AS Código, Sum(Pedidos.Cantidad) AS TOTAL, Avg(Pedidos.Cantidad)

AS Media FROM Pedidos INNER JOIN Artículos ON Pedidos.Id = Artículos.Id

GROUP BY Pedidos.Id, Artículos.Nombre PIVOT Year(Fecha);

y obtenemos el siguiente resultado:

Producto	Código	TOTAL	Media	1996	1997
Zapatatos	1	348	87	300	48
Pantalones	2	955	238,75	375	580
Blusas	3	1940	485	620	1320

Comentarios a la consulta:

La clásula TRANSFORM indica el valor que deseamos visualizar en las columnas que realmente pertenecen a la consulta, en este caso 1996 y 1997, puesto que las demás columnas son opcionales.

SELECT especifica el nombre de las columnas opcionales que deseamos visualizar, en este caso Producto, Código, Total y Media, indicando el nombre del campo que deseamos mostrar en cada columna o el valor de la misma. Si incluimos una función de cálculo el resultado se hará en base a los datos de la fila actual y no al total de los datos.

FROM especifica el origen de los datos. La primera tabla que debe figurar es aquella de donde deseamos extraer los datos, esta tabla debe contener al menos tres campos, uno para los títulos de la fila, otros para los títulos de la columna y otro para calcular el valor de las celdas.

En este caso en concreto se deseaba visualizar el nombre del producto, como el tabla de pedidos sólo figuraba el código del mismo se añadió una nueva columna en la cláusula select llamada Producto que se corresponda con el campo Nombre de la tabla de artículos. Para vincular el código del artículo de la tabla de pedidos con el nombre del misma de la tabla artículos se insertó la cláusula INNER JOIN.

La cláusula GROUP BY especifica el agrupamiento de los registros, contrariamente a los manuales de instrucción esta cláusula no es opcional ya que debe figurar siempre y debemos agrupar los registros por el campo del cual extraemos la información. En este caso existen dos campos del cual extraemos la información: pedidos.cantidad y artículos.nombre, por ellos agrupamos por los campos.

Para finalizar la cláusula PIVOT indica el nombre de las columnas no opcionales, en este caso 1996 y 1997 y como vamos al dato que aparecerá en las columnas, en este caso empleamos el año en que se produjo el pedido, extrayéndolo del campo pedidos.fecha.

Otras posibilidades de fecha de la cláusula PIVOT son las siguientes:

1. Para agrupamiento por Trimestres

PIVOT "Tri " & DatePart("q",[Fecha]);

2. Para agrupamiento por meses (sin tener en cuenta el año)
3. PIVOT Format([Fecha],"mmm") In ("Ene", "Feb", "Mar", "Abr", "May", "Jun", "Jul", "Ago", "Sep", "Oct", "Nov", "Dic");

4. Para agrupar por días

PIVOT Format([Fecha],"Short Date");